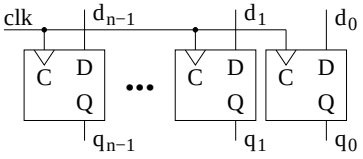
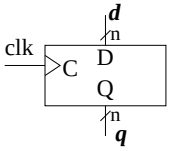
10 Registers and counters

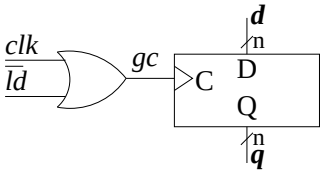
10.1 Registers



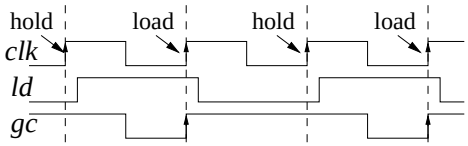
- The simplest n-bit register is a collection of n D flip-flops triggered by a common clock.
- I would prefer to call it just an n-bit D flip-flop, since it performs only a **load** operation on each rising edge of the clock.



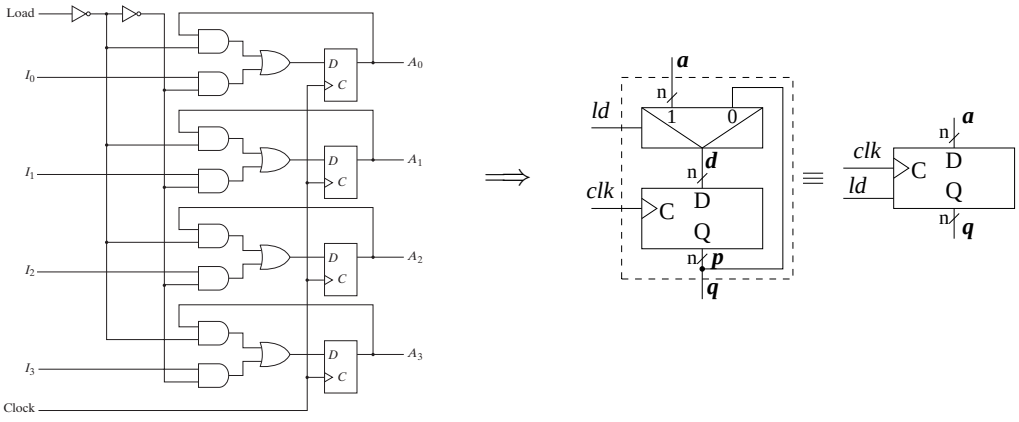
gated clock:



- The simplest n-bit register should perform at least two operations **load** and **hold**
- This can be implemented in two ways:
  - blocking/gating the clock (typically a bad idea for a number of reasons)
  - adding an input multiplexer.



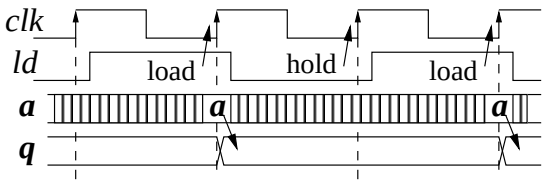
10.1.1 An n-bit "parallel load" register



The operation table of the n-bit register with parallel load:

load	operation	
0	q <= q	hold
1	q <= a	load

The timing diagram:



The VHDL architecture consists of three concurrent statements describing the input multiplexer, the flip-flop process and assignment of internal outputs from the flip-flops to the output port:

```
 LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 USE ieee.std_logic_arith.all;

ENTITY plReg IS
   GENERIC ( N : integer :=4 ) ;
   PORT ( clk, ld : IN  std_logic ;
          A : IN  std_logic_vector(N-1 downto 0) ;
          Q : OUT std_logic_vector(N-1 downto 0) ) ;
END ENTITY plReg;

ARCHITECTURE  rtl  OF  plReg  IS
   SIGNAL P, D : std_logic_vector(A'RANGE) ;

 BEGIN
 -- the input multiplexer:
    D <= A WHEN ld = '1' ELSE  P ;

 -- internal signal on output port
    Q <= P ;

 --  the flip-flop process
    PROCESS (clk)
    BEGIN
      IF clk'EVENT AND clk='1' THEN
        P <= D ;
      END IF ;
    END PROCESS ;
 END rtl ;
```

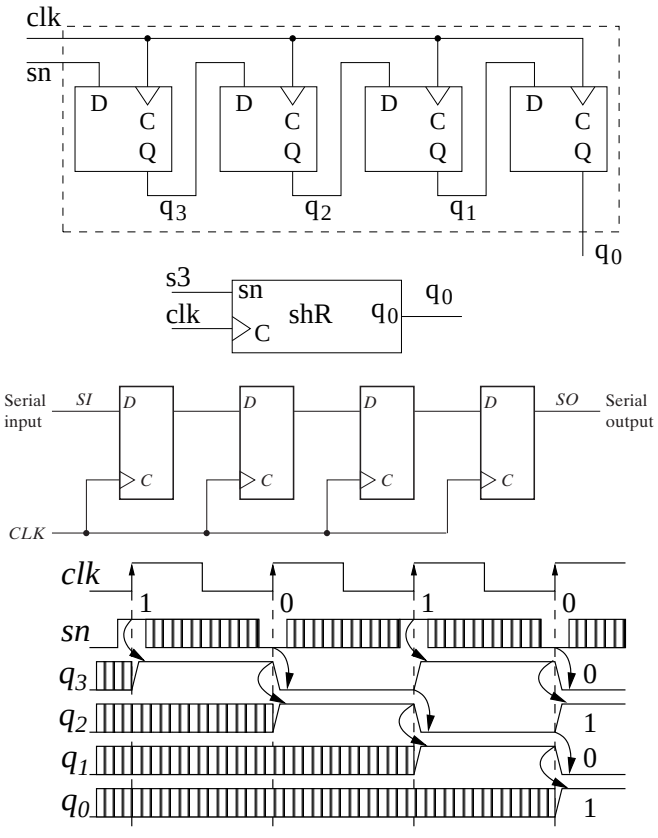

10.1.2 A simple shift register

- In general, a shift register allows the binary words stored in the register to be shifted left or right by one position, with additional bit being shifted in.
- The simplest shift register performs only one operation, say **shift-right**, at each rising edge of the clock, that can be described as (4-bit register has been assumed):

$$\mathbf{q} <= \text{shr}(sn, \mathbf{q}) \,, \quad \text{or}$$

$$(q_3, q_2, q_1, q_0) <= (sn, q_3, g_2, q_1) \,, \quad \text{or}$$

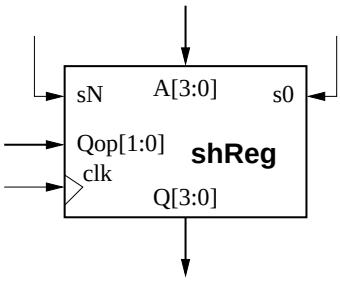
$$q[3:0] <= (sn, q[3:1])$$



- The timing diagram illustrate how a 4-bit binary word (0, 1, 0, 1) presented bit-by-bit at the serial input sn has been shifted into the register during the four consecutive rising edges of the clock.
- Such an operation can be referred to as a "serial load".

10.1.3 A bi-directional shift register

- A bidirectional shift register can be considered as the most **typical sequential component**.
- The register performs both **shift-right** and **shift-left** operations and in addition **load** and **hold** operations.
- Operation to be performed is selected by a 2-bit opcode word.
- Note that we have three types of signals:
  - data (e.g. A, Q),
  - control signals, (Qop), and
  - a synchronizing (clocking) signal (clock).

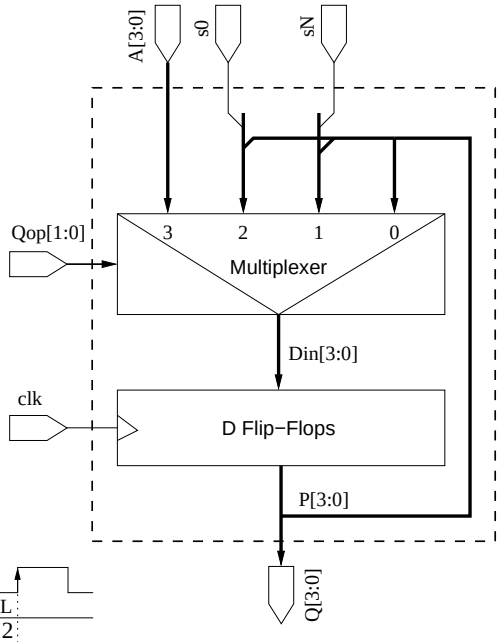


- Specification of the register is given in the form of the following function table (4-bit structure is assumed):
- As usual, operations are performed on the rising edge of the clock.
- Note that there are two single-bit serial inputs, sN and s0 from which the bits are shifted in on the vacated position during the respective shift operation.

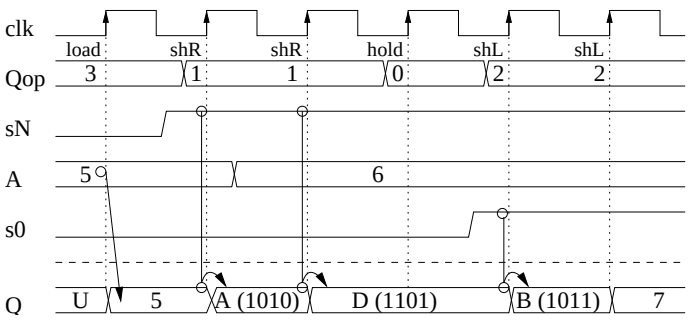
Shift register, Q		
Qop	operation	
0	Q ⇐ Q	hold
1	Q ⇐ (sN, Q[3:1])	shiftR
2	Q ⇐ (Q[2:0], s0)	shiftL
3	Q ⇐ A	load

- The internal structure of the register consists of two main parts:
  - a set of $N = 4$ edge-triggered D flip-flops with outputs P identical to the port signals Q, and inputs, Din.

    The flip-flops ensure the positive-edge sensitivity,
  - a 4-bit 4-to-1 input multiplexer which effectively implements operations as specified in the function table, selecting appropriate signals to be loaded into the flip-flops.



- Timing diagram:



- VHDL code follows from the specification given in the block-diagram and the operation table.
- Two version of the code differs in the way the multiplexer is specified.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all;
ENTITY  shreg  IS
  GENERIC ( N : integer := 4 ; M : integer := 2 ) ;
  PORT ( clk, sN, s0 : IN  std_logic ;
         Qop : IN  std_logic_vector(M-1 downto 0) ;
           A : IN  std_logic_vector(N-1 downto 0) ;
           Q : OUT std_logic_vector(N-1 downto 0) ) ;
END shreg ;
```

```
ARCHITECTURE  rtlA  OF  shreg  IS
  SIGNAL P, Din : std_logic_vector(A'RANGE) ;
  CONSTANT  nop : std_logic_vector(Qop'RANGE) := "00" ;
  CONSTANT  shr : std_logic_vector(Qop'RANGE) := "01" ;
  CONSTANT  shl : std_logic_vector(Qop'RANGE) := "10" ;
  CONSTANT  ldd : std_logic_vector(Qop'RANGE) := "11" ;
BEGIN
   --- Qop range must be static eg.  (1 downto 0)
   WITH  Qop  SELECT
   Din <= P                          WHEN  nop ,
          sN & P(P'LEFT downto 1)    WHEN  shR ,
          P(P'LEFT-1 downto 0) & s0  WHEN  shL ,
          A    WHEN  OTHERS ;

      --      flip-flop process
      PROCESS (clk)
      BEGIN
        IF clk'EVENT AND clk='1' THEN
          P <= Din ;
        END IF ;
      END PROCESS ;
      Q <= P ;
END rtlA ;      END rtlB ;
```

```
ARCHITECTURE  rtlB  OF  shreg  IS
  SIGNAL P, Din : std_logic_vector(A'RANGE) ;
  TYPE arrvec IS ARRAY (natural range <>)
                 OF std_logic_vector(A'RANGE) ;
  SIGNAL YMUX : arrvec(0 to 2**M-1) ;
BEGIN
  YMUX <= ( P,
            sN & P(P'LEFT downto 1),
            P(P'LEFT-1 downto 0) & s0,
            A );
   Din <= YMUX(conv_integer(unsigned(Qop))) ;
```


- Both codes are similar and consists of three concurrent statements: one for a multiplexer, one for flip-flops P and the one which assigns internal signal P to an output port signal Q.
- In the **rtlA** architecture we have specified mnemonic names of constants, which increases code readability.
- The selected signal assignment expression is a bit more limited because the select signal size must be static.
- In the **rtlB** architecture, the multiplexer is specified as an array (table) of 2^m n-bit words.
- The words in the array are equivalent to the multiplexer inputs.
- The opcode Qop selects the n-bit word from the array

Figure 6-4: Serial Transfer from Register A to Register B from Mano

Consider bi-directional serial transfer of data

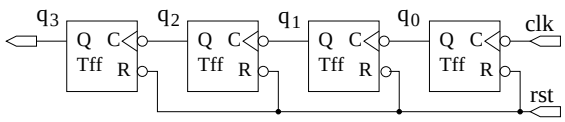
Discuss tri-state line driver.

10.2 Counters

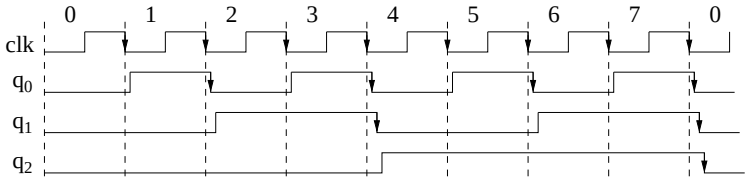
- Counters are sequential circuits that increment or decrement a binary number stored in the flip-flops in response to the rising edge of the clock.
- The name counter is used rather than incrementer/decrementer becuse in the first application of the counters was counting the number of pulses coming to its clock input.

10.2.1 A ripple counter

- The simplest counter, known as a ripple counter, is build from simplified T flip-flops having only the clock input, that is, the toggle input is always on, $T = 1$.

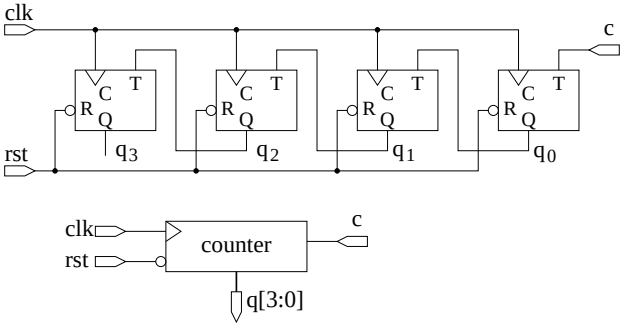


- Note that the flip-flops toggle on the falling edge of the clock.
- In addition the reset signal rst sets the initial stage of the flip-flops to $q = (0000)$
- Timing diagram demonstrate the delay problem associated with the ripple counter.
  The change of the states does not occur strictly on the clock edge, but there is a growing delay between stages.

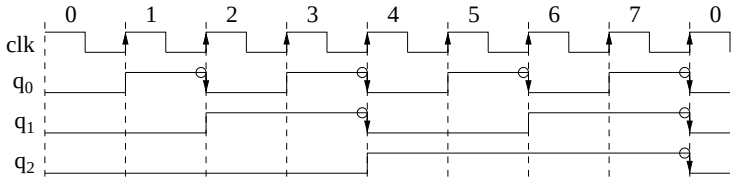


10.2.2 Synchronous counter

- The basic synchronous counter is an improvement on the ripple counter and is typically build from the standard T flip-flops
- The q_i flip-flop is toggled only when the previous flip-flop $q_{i-1} = 1$
- All flip-flops toggle synchronously on the rising edge of the clock.



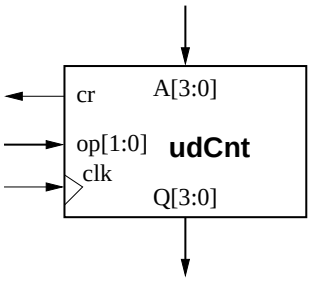
- Timing diagram:



- There are three feature that can be added to the above counter:
  - We might want to start counting from a set number rather than from zero
  - A signal that indicates that the final stage $q = 2^n - 1$ has been reached.
  - We might want to count both up and down.

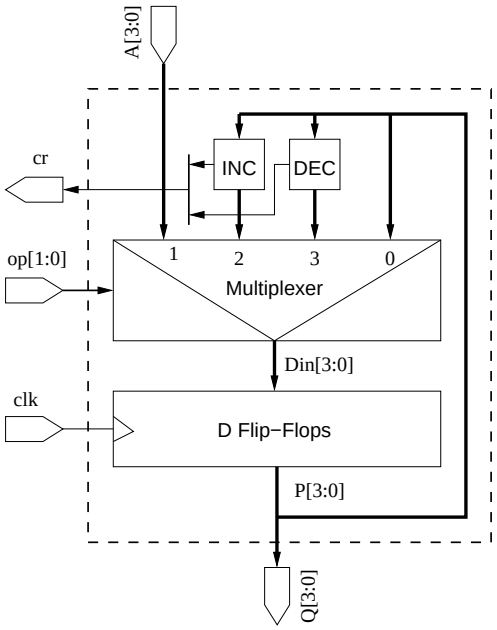
10.2.3 Universal up-down counter

- The universal up-down counter performs four operations **hold**, **load**, **count up** (increment) and **count down** (decrement)
- Operations are selected by a 2-bit opcode word op[1:0].
- In addition a signal cr indicated the maximum (all ones) or minimum (all zeroes) counter contents depending on the direction of counting.
- The above description is formalized by the following operation table:
- As usual, operations ar performed on the rising edge of the clock.
- For the hold and load operations cr has a "don't care" value.



Up-down counter Q			
op	operation		cr
0	Q ⇐ Q	hold	–
1	Q ⇐ A	load	–
2	Q ⇐ Q + 1	inc	Q = max
3	Q ⇐ Q − 1	dcr	Q = min

- Such a universal counter is implemented using D flip-flops to store data and an appropriate excitation circuit.
- The first implementation of the excitation circuit can consist of a 4-to-1 n-bit multiplexer proceeded by combinational incrementer and decrementer.



- A possible VHDL description closely follows the block diagram
- In order to be able to use a simple arithmetic statements to describe increment/decrement operations, the relevant signals are specified as being of the unsigned type
- Fort the unsigned signals we can write statements like

$$Y <= P \pm 1$$

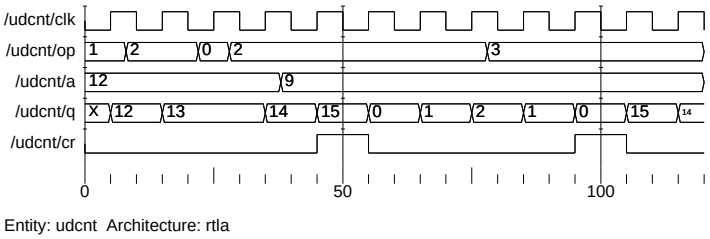
- Implementation of the multiplexer is identical to that discussed for the universal shift register.
- Note that the architecture rtla does not specify details of the implementation of the increment/decrement circuits leaving these details to the synthesizer to decide.

```
--            app,  up-down counter
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all;
ENTITY  udCnt  IS
  GENERIC ( N : integer := 4 ; M : integer := 2) ;
  PORT ( clk : IN  std_logic ;
         op : IN  std_logic_vector(M-1 downto 0) ;
          A : IN  std_logic_vector(N-1 downto 0) ;
          Q : OUT std_logic_vector(N-1 downto 0) ;
         cr : OUT std_logic ) ;
END udCnt ;
ARCHITECTURE  rtla  OF  udCnt  IS
  SIGNAL AA, P, Din : unsigned(A'RANGE) ;
  TYPE  arrvec  IS ARRAY (natural range <>)
                    OF unsigned(A'RANGE) ;
  SIGNAL YMUX : arrvec(0 to 2**M-1) ;
BEGIN
    AA <= unsigned(A) ;
  YMUX <= (P, AA, P + 1,  P - 1) ;  -- multiplexer
   Din <= YMUX(conv_integer(unsigned(op))) ;
    cr <= '1'  WHEN ((op(0) = '0') AND (P = 2**N-1))
               OR ((op(0) = '1') AND (P = 0))
               ELSE '0' ;

   --  flip-flop process
   PROCESS
   BEGIN
     WAIT UNTIL clk'EVENT AND clk='1' ;
     P <= Din ;
   END PROCESS ;

   Q <= std_logic_vector(P) ;
END rtla ;
```

- Simulation waveforms for the universal up-down counter are shown below:



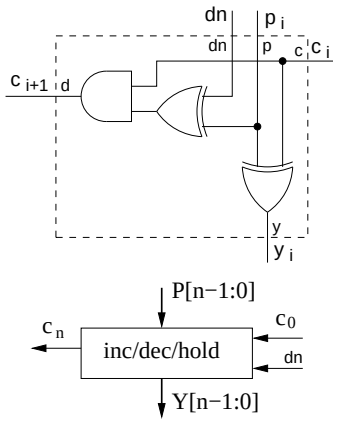
- Inspect the waveforms and verify that all operations are performed as specified in the counter operation table.

- In this architecture of the up-down counter we specify details of implementation of the incrementer/decrementer circuit.
- Following considerations from sec. 7.2 we observe that input and output signals of the incrementer and decrementer are related through the following arithmetic equalities:

$$\begin{aligned} \text{Incrementer:} \quad & c + p = 2 \cdot d + y \\ \text{Decrementer:} \quad & -c + p = -2 \cdot d + y \end{aligned}$$

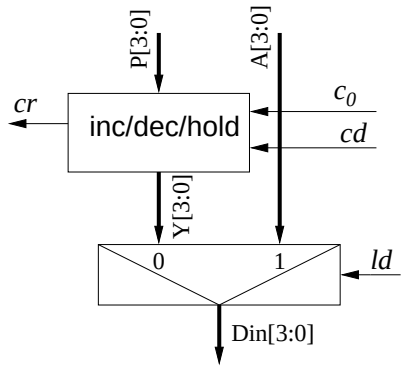
- If we denote by dn a signal to count down, then the arithmetic equalities result in the following logic equations:

$$\begin{aligned} y &= p \oplus c \\ d &= c \cdot (dn \oplus p) \end{aligned}$$



- The 1-bit increment/decrement component can be now connected into an n-bit component as discussed in sec. 7.2.
- The initial carry c_0 must be set up to 1 for increment and to 0 for decrement operation. Otherwise the output will be equal to input.
- We can use this property to implement the hold operation.

- If we use the following increment/decrement circuit, the excitation circuits for the universal up-down counter can be much simplified:
- From the following truth table we can specify the required control signals:



op	op_1	op_0	cd	c_0
hold	0	0	0	0
load	0	1	–	–
inc	1	0	0	1
dec	1	1	1	0

$$cd = op_0$$
$$c_0 = op_1 \cdot \overline{op}_0$$
$$ld = \overline{op}_1 \cdot op_0$$

- The modified VHDL architecture can be written in the following way (D flip-flops has been omitted):

```
ARCHITECTURE  rtlb  OF  udCnt  IS
  SIGNAL P, Y, Din : std_logic_vector(A'RANGE) ;
  SIGNAL      c : std_logic_vector(N downto 0) ;
BEGIN
  c(0) <= op(1) ;

  gnrt: FOR  i  IN  0 TO  N-1  GENERATE -- INC/DEC
      Y(i) <=  P(i) XOR c(i) ;
    c(i+1) <=  c(i) AND (op(0) XOR P(i))  ;
  END GENERATE gnrt ;

  cr <= c(N) ;
  Din <= A  WHEN  op = "01"  ELSE Y ;

   --  flip-flop process

END rtlb ;
```